Microcontrollers

2IN60: Real-time Architectures
(for automotive systems)

Mike Holenderski, m.holenderski@tue.nl

Technische Universiteit Eindhoven University of Technology

Goals for this slide set

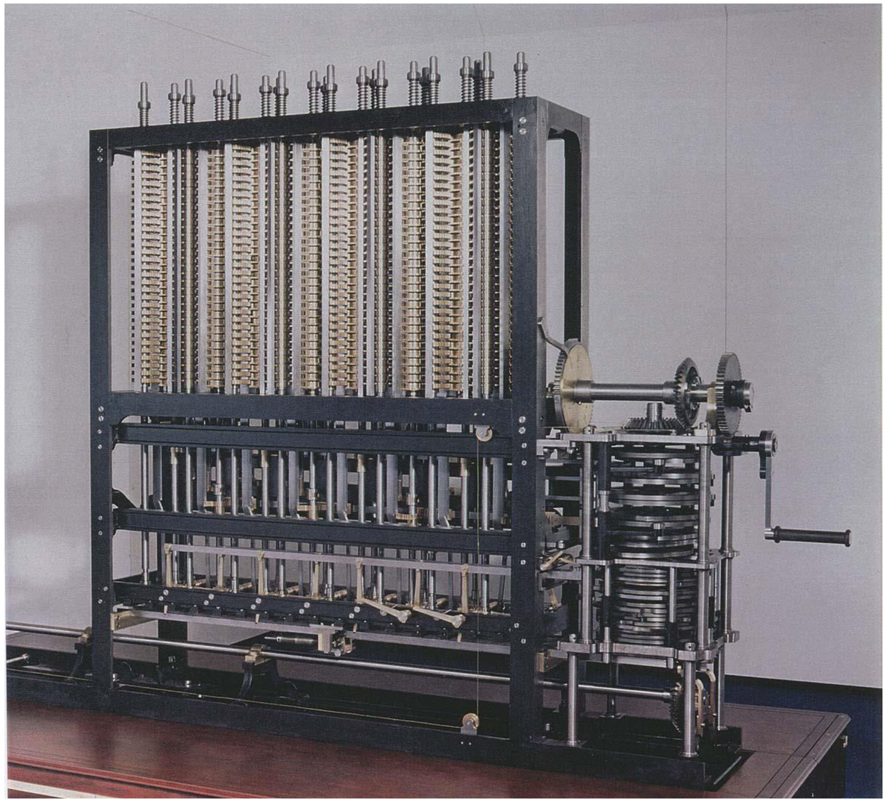
- Describe the architecture of a microcontroller
- Explain the purpose of an Instruction Set Architecture and what it is
- Summarize how programs are executed on a microcontroller
- Describe how programs can communicate with external devices

Outline

- Brief history of microcontrollers
- Example of an ECU
- Instruction set architecture (ISA)
- Input and output (I/O)
- Example

Why electronics?

- Difference Engine No. 2 (1849)
  - Designed by C. Babbage
  - 4000 moving parts
  - 2.6 tons
  - 3m wide, 2m high
- Easier to move information than physical objects

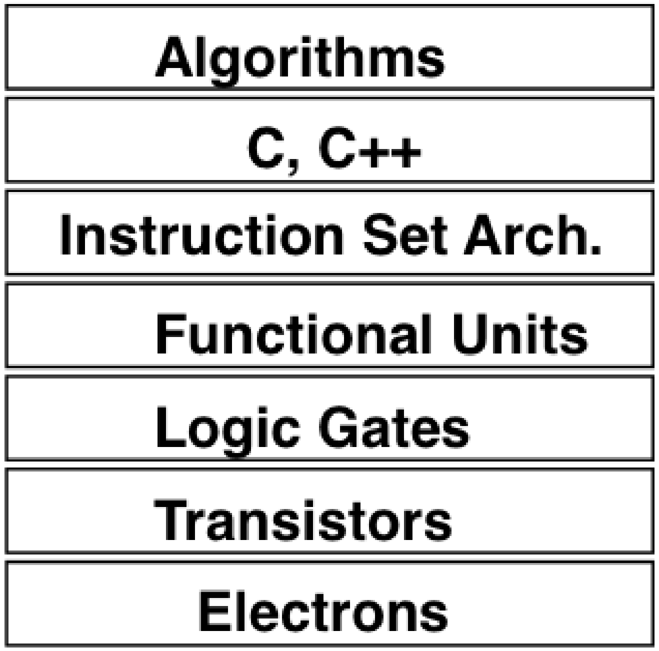


From hardware to software

- Evolution of hardware
  - Tubes → transistors → integrated circuits
- Moore's law:
  - Transistor density doubles every 1.5 years
- Problem: **complexity**
  - Large number of applications, performing increasingly complex tasks
  - Can't afford dedicated hardware for individual apps
    - too many apps, large design cost
  - Multiplex several applications on the same hardware
- Solution: **reuse general-purpose components**
  - Generic controllers customized with memory (hardware)
  - Memory contains a program (software)
  - Stored-program concept (code + data in memory)
- Example: Fuel injection (see Introductory slides)

Fighting complexity

- Use abstraction to hide complexity, while keeping the essentials
  - Define an interface to allow people to use features without needing to understand all the implementation details
- Works for hardware and software
- Stable interface allows people to optimize below and above it



Why should I care how a computer works?

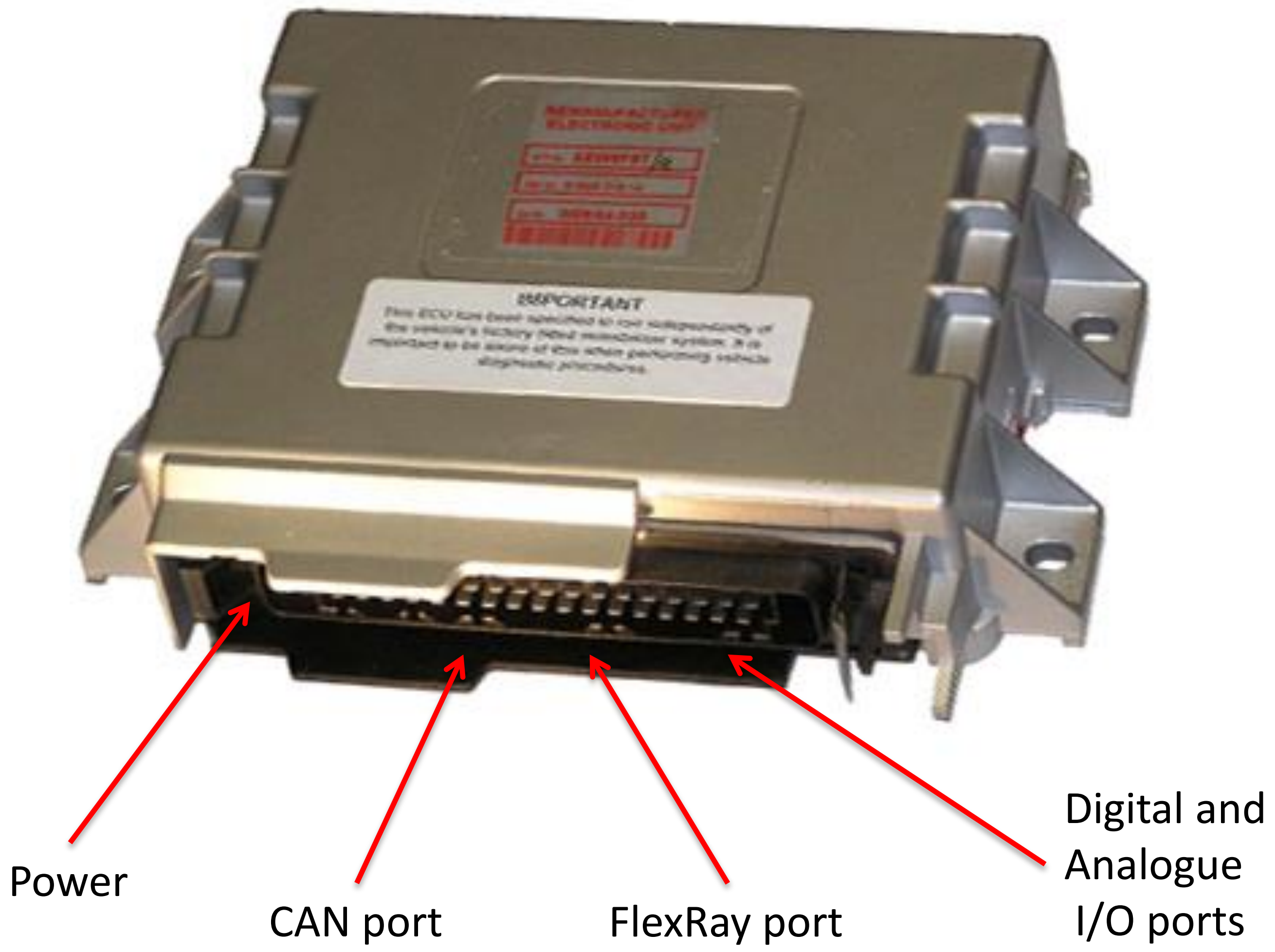
- Understand the limitations of hardware and its abstractions
- Understand the performance bottlenecks
- Help write better programs (fewer bugs, more efficient)

- Understanding the machine-level execution model is critical for:
  - Understanding the source of common challenges, such as race conditions in concurrent systems
  - Understanding behavior of programs in the presence of bugs (when higher-level models brake down)
  - Tuning program performance (understanding sources of program inefficiency)
  - Writing critical code still programed in assembly

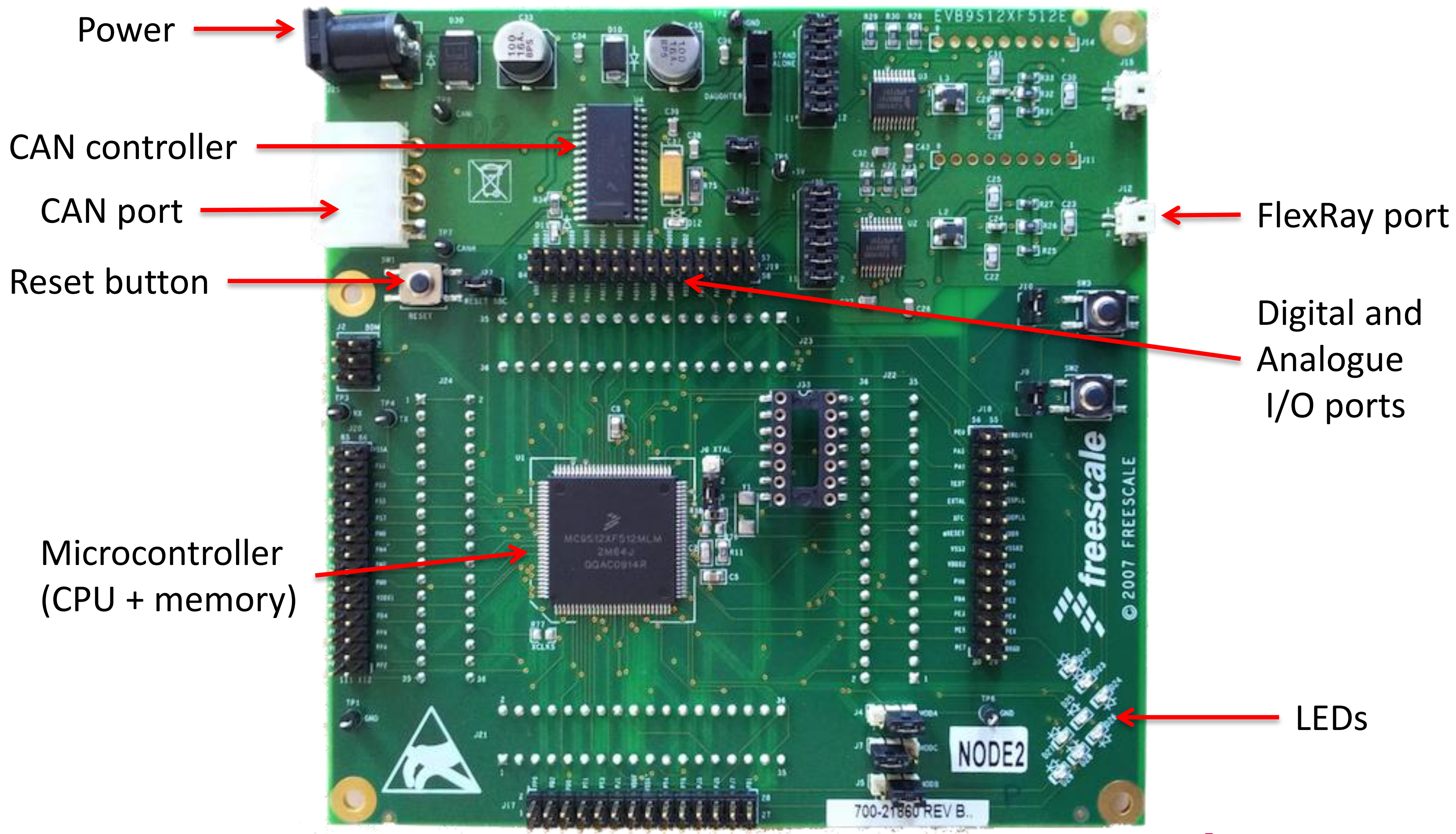
Outline

- Brief history of microcontrollers
- **Example of an ECU**
- Instruction set architecture (ISA)
- Input and output (I/O)
- Example

An ECU and its interfaces



Example ECU (Freescale board EVB9512XF)

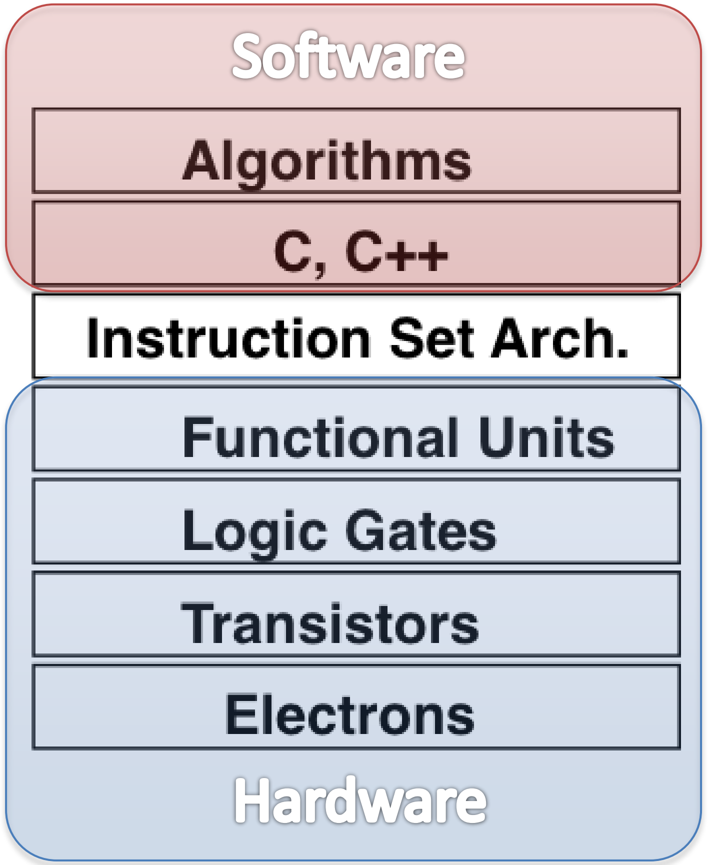


Outline

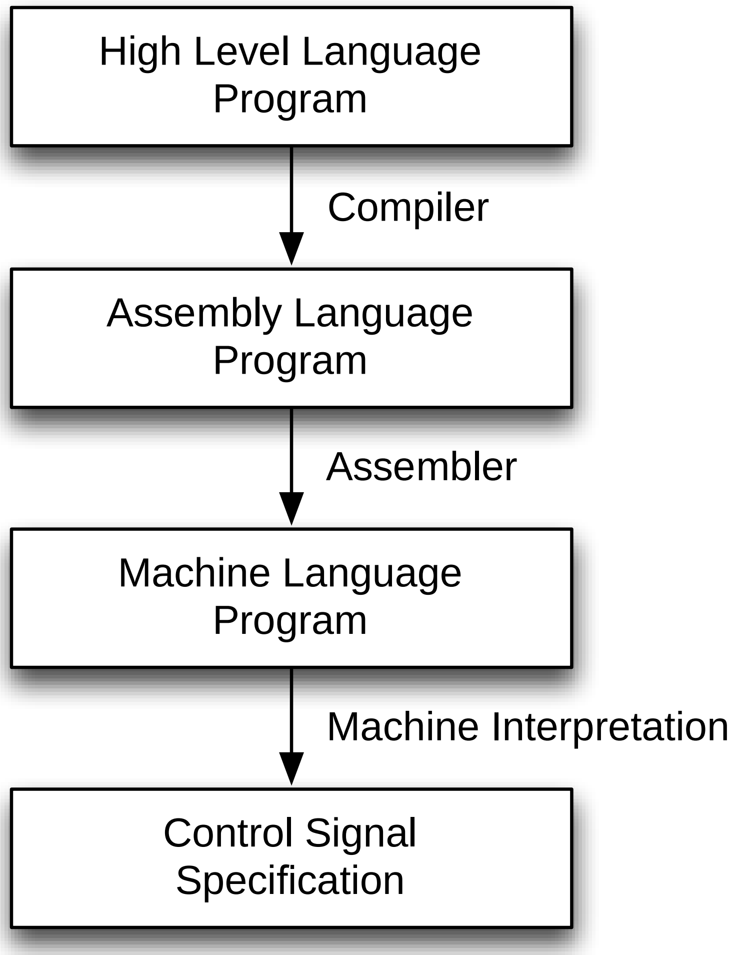
- Brief history of microcontrollers
- Example of an ECU
- **Instruction set architecture (ISA)**
- Input and output (I/O)
- Example

Instruction Set Architecture (ISA)

- CPUs only work with binary signals (bits)
- Machine instructions: strings of bits telling hardware what to do
- Machine code: sequence of machine instructions
- ISA is the interface between hardware and software
  - Maps between program text and bits
  - Defines what instructions do
  - Syntax defined by assembly language
    - Symbolic representation of the machine instructions
  - Examples: MIPS, x86, PowerPC, ARM



Running an application


High Level Language Program
Compiler
Assembly Language Program
Assembler
Machine Language Program
Machine Interpretation
Control Signal Specification

```
temp = a[0];
a[0] = a[1];
a[1] = temp;
```

```
LDD   6,-SP          temp = a[0];
STD   4,SP
LDX   2,SP           a[0] = a[1];
STX   0,SP
STD   2,SP           a[1] = temp;
```

```
11101100 10101010
01101100 10000100
11101100 10000010
01101100 10000000
01101100 10000010
```

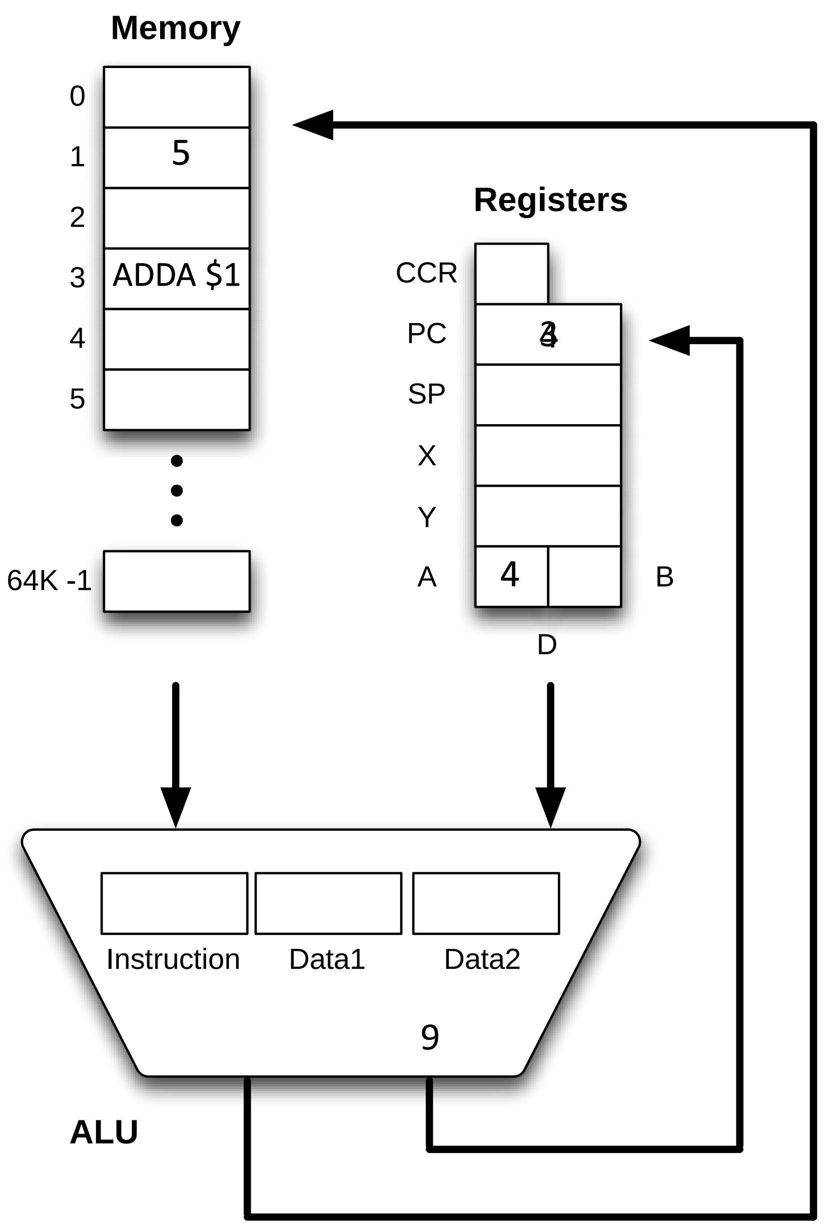
High/Low on control lines

Central Processing Unit (CPU)

- Control unit (CU) organizes everything, i.e.
  - fetching, decoding and executing instructions: the so-called “fetch-decode-execute” cycle
- Arithmetic logical unit (ALU) performs operations to carry out the instructions
- Registers are fast (and small) memory
  - “program counter” (PC): points to the next instruction to be executed
  - general/special registers: to store temporary results

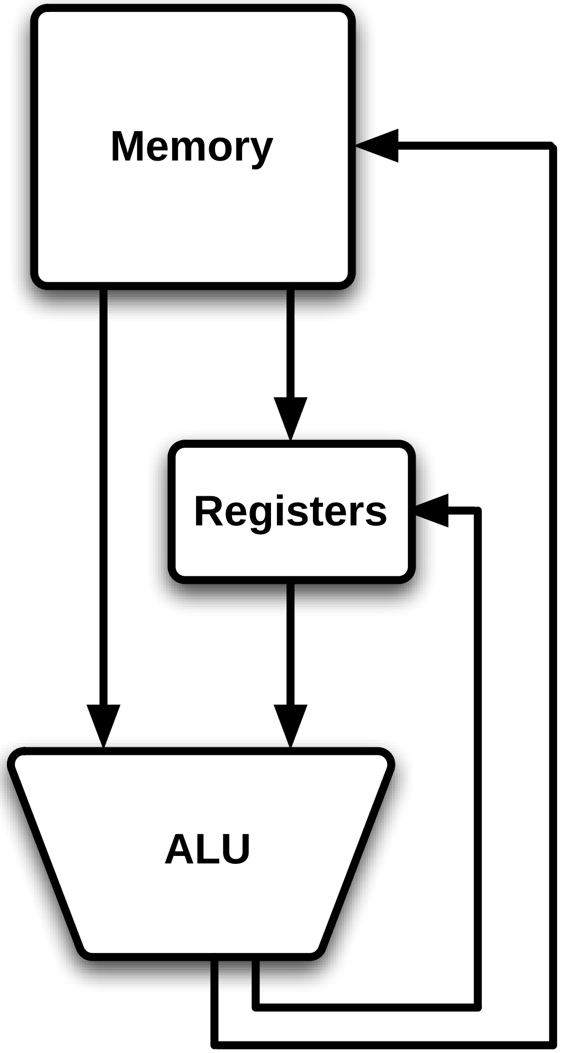
Instruction execution

1. Fetch the instruction from memory (pointed to by PC)
2. Fetch the data from memory into internal CPU registers (specified by instruction operands)
3. Execute the instruction
4. Store the results at the proper place (memory or registers)
5. Change PC so that it points to the following instruction
6. Go to step 1 to begin executing the following instruction.



Load-store ISA

- Why not operate directly on memory operands?
  - Accessing memory is slower than CPU registers, so instructions would take longer to execute
- Load-store ISA
  - Data is loaded into registers, operated on, and stored back to memory or registers
- Freescale HCS12 ISA
  - Arithmetic operators can use register *and* memory operands
    - E.g. ADDD $20
      - Add the value at address 20 to the value in register D and store the result in D



Assembly instructions (Freescale HCS12)

- Assembly instruction format:
  - <operation name> <operand1>, <operand2>
- Examples
  - MOVB $20, PC
    - Move the byte at memory location 20 to PC
  - ADDD $20
    - Add the two bytes at memory location 20 to register D, and store in D
  - ABA
    - Add the byte in register B to the one in A, and store in A
- Break larger expressions into multiple instructions

a = b + c – d; ⇒

```
LDD b
ADDD c
SUBD d
```

  - (Result 'a' will reside in the D register)

Example ISA: Freescale HCS12

	C	Freescale
Registers		program counter (PC), stack pointer (SP), condition code register (CCR), accumulators (A,B), index registers (X,Y)
Memory	local variables global variables	Linear array with 64KB
Data types	int, char, float, unsigned, struct, pointer	byte (8b), word (16b)
Operators	+, -, *, /, %, ++, <, etc.	ABA, ADDA, SBA, SUBA, MUL, EDIV, INCA, CMPA, ANDA
Memory access	a, *a, a[i], a[i][j]	LDS, LDAA, LDAB, STS, STAA, STAB, MOVB, MOVW, TFR, PSHA, PULA
Control	If-then-else, while, procedure call, return	BEQ, BLE, BLT, LBEQ, LBLE, LBLT, DBEQ, JMP, JSR, RTS, SWI, RTI

Outline

- Brief history of microcontrollers
- Example of an ECU
- Instruction set architecture (ISA)
- **Input and output (I/O)**
- Example

Input / Output (I/O)

- “Device” is a physical “thing” in the outside world
  - “Outside world” = outside CPU and memory
  - E.g. a pressure sensor or brake connected via an analogue port, or another ECU connected via CAN port
- “Controller” connects device and CPU
  - Electrical conversion and (“intelligent”) control
  - E.g. the CAN controller
- “Interface” is an agreement for the connection
  - Physical (plug!), electrical and functional (protocol)
  - Standardized for exchangeability

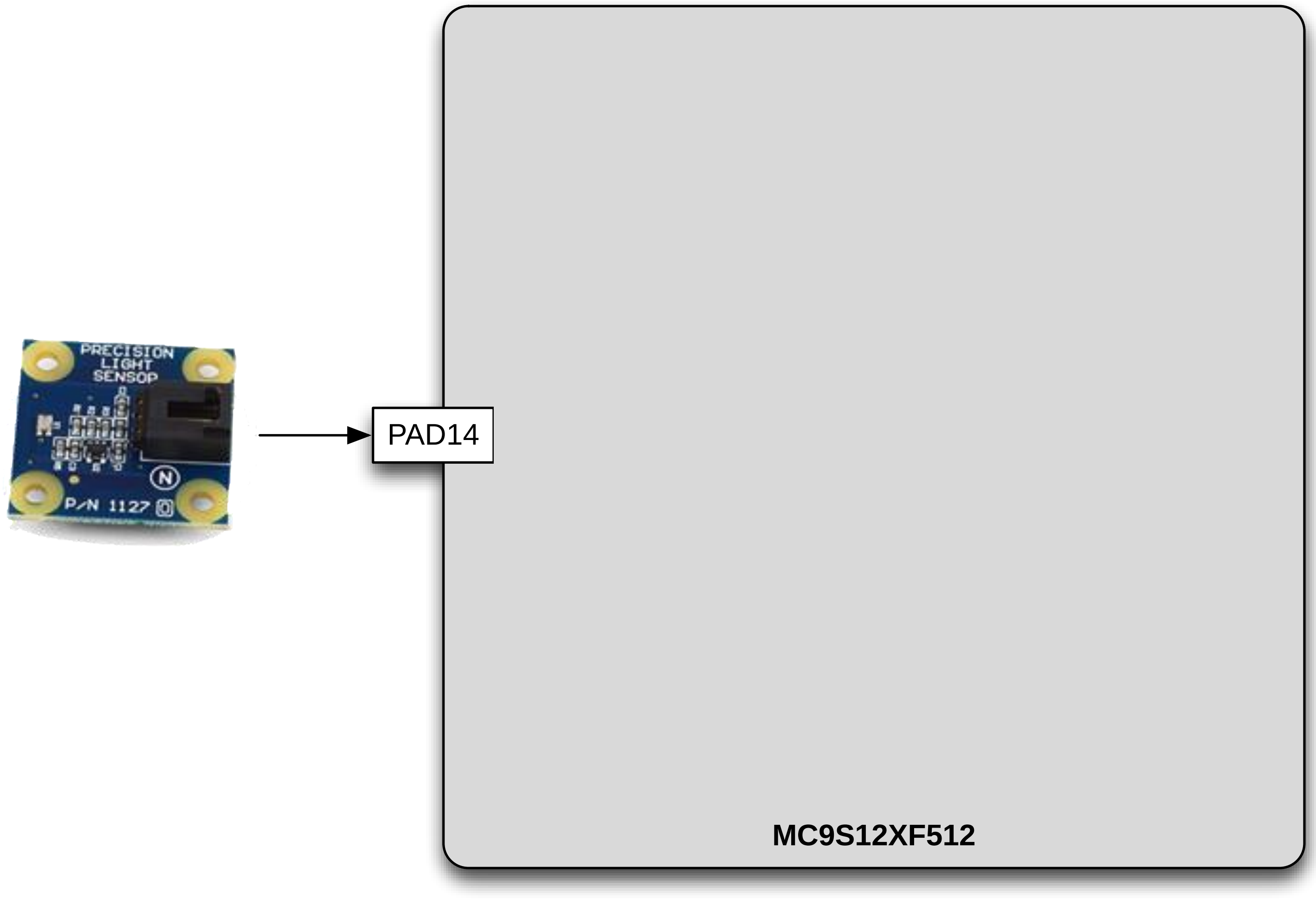
Input / Output (I/O)

- Program can communicate with external devices by reading and writing data to the input and output ports
  - Digital devices are connected to the digital ports
  - Analogue devices are connected via an Analogue-to-Digital (ATD) converter
    - Converts voltage to a discrete value
  - Each port is mapped to a memory address
  - Some devices need to be setup and controlled by writing to special memory addresses (control registers)
- Embedded FlexRay and CAN microcontrollers provide higher level communication between several processing boards

Outline

- Brief history of microcontrollers
- Example of an ECU
- Instruction set architecture (ISA)
- Input and output (I/O)
- **Example**

Example: reading a light sensor (Freescale MC9S12XF512)

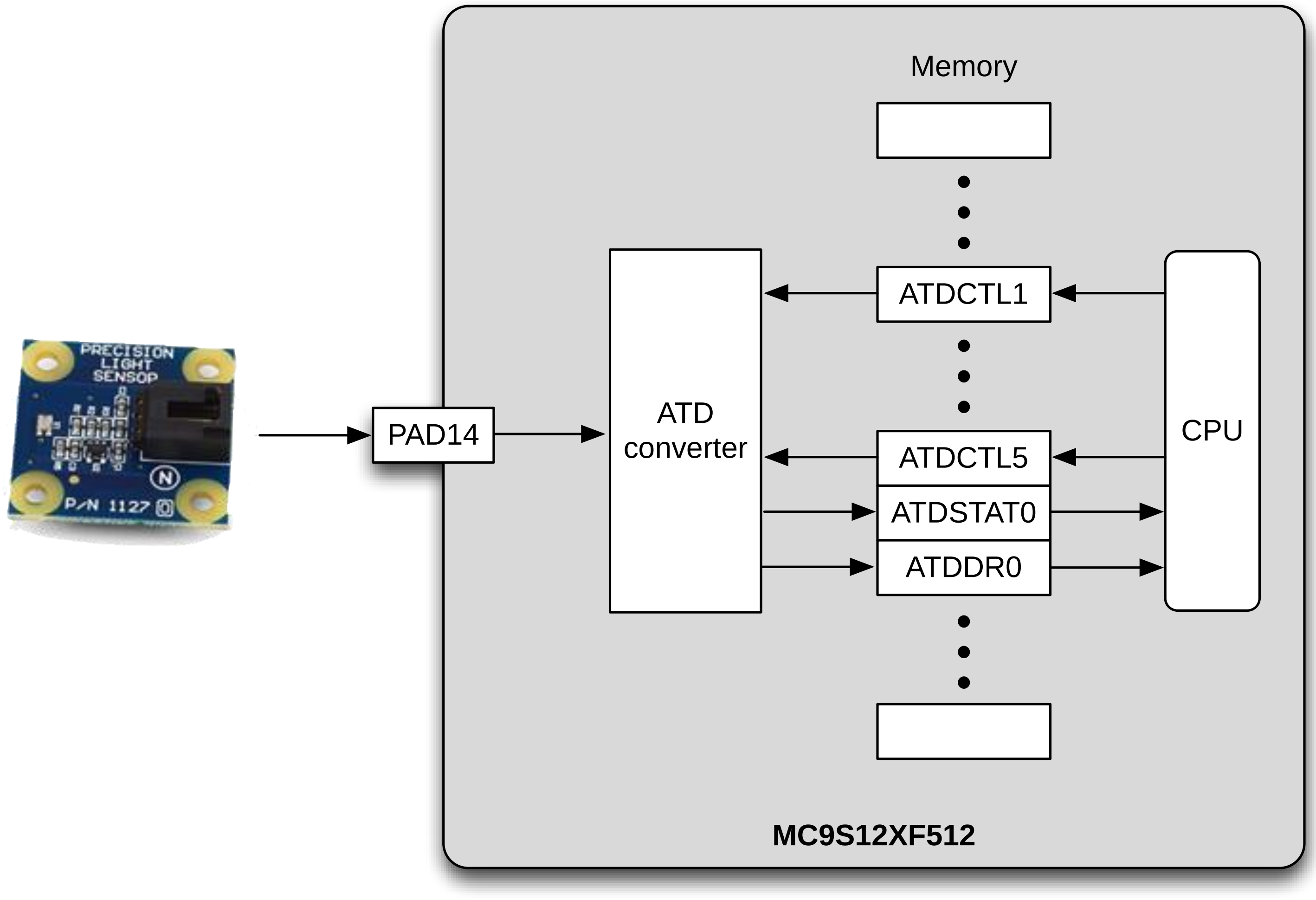


Example: reading a light sensor

- We want to read the light sensor
- The light sensor is connected via an ATD converter on the I/O port `PAD14`
- We need an ATD driver, providing a method which can be called from a C program, e.g.

```
int light = ATDReadChannel(PAD14);
```

Example: reading a light sensor (Freescale MC9S12XF512)



Example: reading a light sensor (Freescale MC9S12XF512)

- Implementation of **`ATDReadChannel()`**:
  - Conversion parameters (controlled by registers ATDCTL1 – ATDCTL5):
    - Sample resolution (8, 10, or 12 bits)
    - Sample duration (44 – 2688 cycles)
    - Number of consecutive samples
    - …
  - Conversion is started by writing to ATDCTL5
  - When conversion is finished a bit in ATDSTAT0 is set
  - Results are stored in registers ATDDR0 – ATDDR15

Example: reading a light sensor (Freescale MC9S12XF512)

`MOVB #$10,ATDCTL1`	set A/D resolution to 8 bit
`MOVB #$88,ATDCTL3`	set conversion sequence length to 1
`MOVB #$05,ATDCTL4`	set sampling time to 264 cycles
`MOVB PAD14,ATDCTL5`	set which channel to read
`SCF: BRCLR ATDSTAT0,#$80,SCF`	wait for the result
`BSET ATDSTAT0,#$80`	clear the ATD result flag
`LDX ATDDR0`	load the result into register X

References

- D.Patterson and J. Hennessy, “Computer Organization & Design”, 3rd Edition, 2005
- Niklaus Wirth, “Algorithms and Data Structures”, Prentice-Hall, 1985